Prototyping DSU techniques using Python

Sébastien Martinez, Fabien Dagnat, Jérémy Buisson

Prototyping DSU techniques using Python

Sébastien Martinez and Fabien Dagnat
Université Européenne de Bretagne
IRISA / Institut Mines-Télécom / Télécom Bretagne

Jérémy Buisson
Université Européenne de Bretagne
IRISA / Écoles de St-Cyr Coëtquidan

Abstract

This paper presents PyMoult, a Python library implementing various dynamic software update (DSU) mechanisms. This library aims to provide a prototyping platform for experimenting with DSU and to implement a vast choice of update mechanisms while allowing their combination and customization.

We selected different update mechanisms from the literature and implemented them in PyMoult. This paper focuses on how we implemented these mechanisms and discusses the cost of implementing DSU in Python.

1 Introduction

A huge number of *Dynamic Software Update* platforms have been proposed. For instance, Seifzadeh et al. [8] cite around fifty proposals. Several surveys [5, 7, 8, 9] help a software engineer needing such a platform by comparing their characteristics and constraints. Either an existing DSU platform fits exactly his requirements or he has to come up with a new ad-hoc solution. Indeed each proposal presents a platform, not a collection of reusable and composable mechanisms.

By implementing several update mechanisms inspired by existing DSU platforms, PyMoult is addressed to different kinds of users: a DSU researcher experimenting new mechanisms, a program developer testing a dynamically updatable program or an update developer wanting to test an update prototype. Our intent in PyMoult is to provide customizable and combinable building blocks that can be reused and adapted to implement specific update mechanisms. This paper reports the first version of PyMoult and discusses its implementation.

PyMoult is written in Python because it is a good language to prototype complex applications.

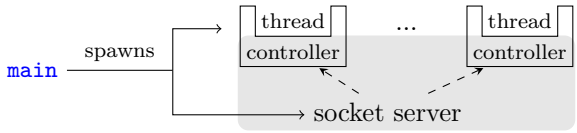

Figure 1: Structure of a PyMoult program.

Being a dynamic language and supporting modules relinking, Python simplifies DSU implementation. We use the PyPy Python interpreter. In comparison to Cpython (the most widely used one), PyPy offers additional functionalities useful to implement DSU, among which continulets (see §2.1). Furthermore, PyPy is written in Python making it easier to modify, if needed. Running the interpreter in an interpreter incurs an overhead; but as we only intend to prototype DSU, the overhead does not matter.

Section 2 presents how to use PyMoult and the update mechanisms currently provided. Section 3 demonstrates the capabilities of PyMoult using an example with several updates. Then Section 4 discusses issues met with PyPy and the choice of Python for our intent. Our concluding remarks and future directions are in Section 5.

2 Update mechanisms

PyMoult controls programs wrapped within an update manager. As shown on Figure 1, an update manager consists of a set of controllers (one for each threads) and a socket server which listens to update commands and forwards them to the controllers.

A user of PyMoult must define the tasks of his application and create a thread for each of them. Each thread may be active or passive depending on the way its controller manages updates as detailled in §2.1. The user must register them to the socket server using the `register_threads` function. This

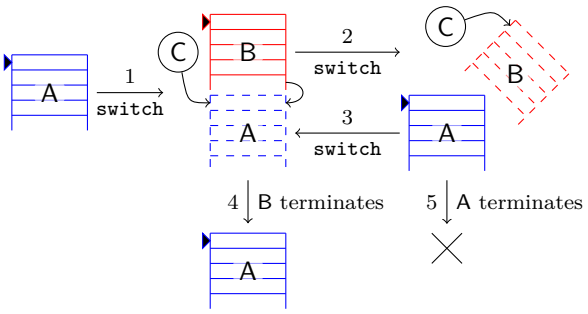

Figure 2: Continulets of PyPy

enables the user to select threads that will not be managed and therefore, not updated.

To update his application, the user must also write a new Python module that configures the update (*e.g.* defines new classes and functions, selects an update mechanism). When receiving an update command, the socket server loads and triggers this update module dynamically into the program.

Threads are the basic unit PyMoult considers. While updates are triggered for the whole program, each thread has an update function and starts it independently. An update function implements all the operations needed for an update such as for example replacing functions, converting objects, ensuring quiescence or starting new threads. PyMoult provides tools to handle the most usual update operations (see §2.2 and §2.3). Notice that as each thread has its own update function, the user may trigger updates targeting only a subset of the threads (which is equivalent to updating only a part of the program). This also means that updates concerning several threads (*e.g.* updating functions called in several threads or shared objects) and all concurrency matters must be handled by the update functions.

2.1 Controlled threads

In Python, a thread cannot pause another thread and access its stack. We use continuations to simulate this feature, using PyPy's continulets.

A continulet is a continuation that can be switched to and from at will and is associated with a function. Figure 2 explains how it works. Let A be the continuation of a thread and C a continulet. (1) When the thread invokes the `switch` method of C, a new continuation B (running the function associated with C) is pushed onto the stack and activated. (2) If the code in B calls the `switch` method of the continulet, it is paused and A resumes. (3) If A invokes `switch` again, B resumes. The program continues switching between A and B until one of the two terminates. (4) If B terminates, A definitively resumes; the continulet cannot be switched anymore. (5) If A terminates before B, the continuation B is cancelled.

Using continulets, we implement two kinds of controllers. Both can pause their running code and update themselves in a new continuation before resuming their execution. An *active thread* (spawned by `start_active_threads`) starts its update immediately after receiving the update trigger. Whereas a *passive thread* (spawned by `start_passive_threads`) starts updates only when its execution reaches some specific points and the update has been triggered[1].

Active threads use the trace facility of the debugging infrastructure: a trace function is called after each line of code. Our trace function checks if an update has been triggered and if so uses a continulet to switch to the controller to run its update function.

In passive threads, the code of the target application must invoke the `start_passive_update` function, which plays the same role as the trace function of active threads. These calls should be placed on spots where the user thinks the program may be quiescent. Checking quiescence is under the responsibility of the update function[2]: if the checking fails the update function will be re-executed at next point.

The following code illustrates thread controllers.

```
def passive_main():
  ...
  start_passive_update()
  ...
def active_main(): ...
threads = start_passive_threads(passive_main)
threads += start_active_threads(active_main)
# Starting the manager with the configured threads
register_threads(threads)
```

2.2 Stack manipulation

In this section, we address how an update function checks the state of its thread in the runtime stack, *e.g.*, to detect quiescence. PyMoult provides functions handling two update mechanisms using stack manipulation. *Safe function redefinition* relinks a function when the stack contains no call to that function. *Thread reboot* stops definitely the function of a thread and starts another function instead.

Safe function update is used by Opus [1] and Ksplice [2]. It walks the stack frame by frame. When it finds a call to the function, the update is aborted and retried later. If the function is not in the stack,

[1] We named these threads *passive* and *active* from the point of view of the manager.

[2] And therefore, of the user.

```
Socket_port = 5678
class Product(object): ...
class Site(object): ...
company = []
def get_site_by_name(name): ...
#Manager active thread
def site_manager_main():
  # Initialization of the sites
  ...
  # Main loop
  while True:
    for site in company:
      print(string_of(site))
    print("")
    for x in range(10):
      time.sleep(1)
def order(quantity,product,site):
  the_site = get_site_by_name(site)
  print(the_site.order(product,quantity))
def do_command(command):
  opers = command.split()
  if opers[0] == "order" and len(opers) == 4:
    order(int(opers[1]),opers[2],opers[3])
def socket_main():
  s = socket.socket(socket.AF_INET,
                    socket.SOCK_STREAM)
  s.bind((socket.gethostname(),Socket_port))
  s.listen(5)
  while True:
    conn,addr = s.accept()
    command = conn.recv(9999)
    start_passive_update()
    do_command(command)
    start_passive_update()
    data = ""
    conn.close()
# Spawn application threads
pool = enable_eager_object_conversion()
threads = \
  start_active_threads(pool,site_manager_main) \
  + start_passive_threads(pool,socket_main)
register_threads(threads)
```

Figure 3: Version 1 of example.py.

it is relinked to its updated version. For example in the code below, `new_fun` redefines safely `fun`.

```
def new_fun(args): ...
upd_fun = safe_redefine("fun",new_fun,"__main__")
set_update_function(upd_fun,thread)
```

Thread reboot (a mechanism of ReCaml [3]) stops the continuation of the thread and replaces it with a new one. The user can extract values from the stack to use them in the new function, for instance to initialize it or to implement *stack reconstruction*. In the following code, the thread restarts with `new_main`.

```
def new_main(args): ...
update_fun = reboot_thread(new_main)
set_update_function(update_fun,thread)
```

2.3 Heap manipulation

This section describes how an update converts objects of given classes in the heap. We implement two mechanisms for this purpose among all the scheduling strategies. *Eager object conversion* updates the objects at update time. *Lazy object conversion* updates the objects when the program needs them.

Eager object conversion uses the same mechanism as DynamicML [4]. Because PyPy does not let us walk the heap, we hook object creation to store weak references in a *global pool*. The update function can access this global pool to relink immediately the classes of the objects. The code below shows how to activate the pool of weak references and how to update the class `Foo` using eager object conversion.

```
# Activating the pool
pool = enable_eager_object_conversion()
threads = \
     start_active_threads(pool,main1,main2) \
   + start_passive_threads(pool,main3,main4)
register_threads(threads)
# Using eager object conversion in thread 0
class FooV2(object): ...
update_fun = eager_update_class(Foo,FooV2)
set_update_function(update_fun,threads[0])
```

Lazy object conversion mimics the mechanism of Ginseng [6]. PyMoult uses Python's attribute access hook to trigger the conversion function. The following code shows how to update the class `Foo` using lazy object conversion.

```
class FooV2(object): ...
start_lazy_update_class(Foo,FooV2)
```

To facilitate the implementation of update functions, PyMoult offers high level functions that update objects of a given class. Theses functions can be used with the two previously presented mechanisms.

Eager and lazy object conversion mechanisms are both compatible with the passive and active controlled threads presented in the previous section.

3 Using an example: Products and site management

To illustrate the previous matters, we use a simple program managing products stored on stock sites.

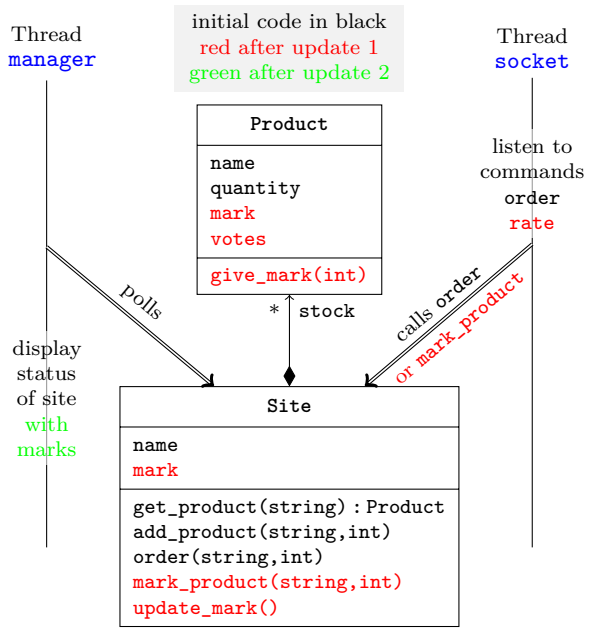

Figure 4: Product and site management

We design two updates, which combine the previously described mechanisms. Figure 4 presents the structure of this example with its updates while extracts of its initial code are in Figure 3. The program is composed of two threads and two classes. The `Product` class is associated to the `Site` class; the (passive) `socket` thread listens for text commands; the (active) `site_manager` thread displays periodically the status of the sites.

The first version of the program only allows to order products from a given site. In the first update (Figure 5), products and sites may additionally be rated. The simultaneous update of the two threads can lead to version synchronisation problems if the `socket` thread is updated before `site_manager`. Although we are supposed to, we do not consider this issue to keep the code short enough. We use safe redefinition to relink the `do_command` function of the `socket` thread. Instances of `Product` are converted lazily while instances of `Site` use eager conversion.

After the second update, the `site_manager` thread will display the ratings of the sites in their statuses (Figure 6). For this, we use thread reboot. This update does not need any value from the old stack.

4 Discussions

4.1 Pypy issues

Some issues with PyPy lead us to use workarounds to implement some mechanisms in PyMoult.

It is not possible in Python to pause a thread and access its stack from another thread. Instead, we run part of the update manager in each thread. As described in Section 2.1 we use continulet to simulate that the update manager is in distinct threads.

Eager object conversion has to access all the objects of the heap. As PyPy does not allow it, we have to maintain a pool of weak references when objects are created. PyPy does not provide any way to intercept object creation. Instead, we intercept calls to functions named `__init__`, which are constructors in Python conventions. Modifying the PyPy interpreter (exposing a heap walker or hooking object creation) would make such assumption unnecessary.

When accessing the frames of a running program, we only access mirror objects. Any modification of the stack is therefore impossible. Offering an access to the real frames could enable mechanisms relying on stack modification.

4.2 Language discussion

As seen in the previous sections, implementing these mechanisms in Python required some efforts and introduced overhead. Using traces to intercept update triggering or object creation disables the JIT. On the other hand, using continulets makes it easier to pause a thread and run its update. Moreover, relinking new functions or classes is free thanks to Python's native indirection. In languages like C, data may be outside of the heap (*e.g.* in the stack). This makes it less natural to implement eager data conversion. It would also require some overhead to keep references to every variables. Moreover, to implement lazy data updates, Ginseng uses preprocessing to insert update code before each variable usage whereas we do not need such a step when using Python. Relinking and redefining classes or functions in Python is free whereas it would request some modifications of the Java Virtual Machine and C would require using indirections.

Thanks to the flexibility of Python, the implementation of all the update mechanisms we tested induced a low development cost.

5 Conclusion

We have presented four mechanisms taken from four different DSU platforms and two ways of starting updates. All these mechanisms can be arbitrarily combined, even within a single application. This paper focuses on the high-level functions of PyMoult but we also provide lower-level functions for users needing to write their own update functions. We have

```
#References to program modules and threads
main = sys.modules["__main__"]
threads = main.threads
manager_thread = threads[0]
socket_thread = threads[1]
#Object updates
class ProductV2(object):
  ...
  def __convert__(self):
    self.mark = 0
    self.votes = 0
  def give_mark(self,mark): ...
start_lazy_update_class(main.Product,ProductV2)
class SiteV2(object):
  ...
  def __convert__(self):
    self.mark = 0
  def mark_product(self,product,mark): ...
  def update_mark(self): ...
site_update = eager_update_class(main.Site,SiteV2)
```

```
#We execute the eager update in the manager thread
set_update_function(site_update,manager_thread)
#Function updates
def rate(mark,product,site):
  the_site = main.get_site_by_name(site)
  print(the_site.mark_product(product,mark))
def new_do_command(command):
  operands = command.split()
  if operands[0] == "order" \
         and len(operands) == 4:
    main.order(int(operands[1]),operands[2],
               operands[3])
  if operands[0] == "rate" \
         and len(operands) == 4:
    rate(float(operands[1]),operands[2],
         operands[3])
socket_update \
    = safe_replace("do_command",new_do_command,
                   "__main__")
set_update_function(socket_update,socket_thread)
```

Figure 5: The update of the example.

```
main = sys.modules["__main__"]
mgr_thread = main.threads[0]
# Manager update
def new_mgr_main():
  company = main.company
  while True:
    for site in company:
      print(string_with_marks_of(site))
    for x in range(10):
      time.sleep(1)
mgr_upd_fun = reboot_thread(new_mgr_main)
set_upd_function(mgr_upd_fun,mgr_thread)
```

Figure 6: Second update of the example

used these functions to reimplement the examples of DynamicML [4] and ReCaml [3] using PyMoult.

Only eager and lazy scheduling strategies are implemented for object conversion, according the philosophy of PyMoult we aim to provide more strategies and to let users write their own strategies. While implementing new mechanisms in PyMoult, we intend adding several parallel functionalities like quiescence detection or correctness checking.

PyMoult is GPL software available on bitbucket.

http://bitbucket.org/smartinezgd/pymoult

Acknowledgments

This work is funded by the Brittany Region council and took place in the Télécom Bretagne and Saint-Cyr Coëtquidan engineering schools.

References

[1] Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. Opus: online patches and updates for security. In *USENIX Security Symposium* (Baltimore, Maryland, USA, Aug. 2005), pp. 287–302.

[2] Arnold, J., and Kaashoek, M. F. Ksplice: automatic rebootless kernel updates. In *European Conference on Computer Systems* (Apr. 2009), pp. 187–198.

[3] Buisson, J., and Dagnat, F. Recaml: execution state as the cornerstone of reconfigurations. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (2010), ICFP '10, pp. 27–38.

[4] Gilmore, S., Kirli, D., and Walton, C. Dynamic ML without dynamic types. Tech. Rep. ECS-LFCS-97-378, The University of Edinburgh, 1997.

[5] Miedes, E., and Muñoz-Escoí, F. D. A survey about dynamic software updating. Tech. Rep. ITI-SIDI-2012/003, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, May 2012.

[6] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical dynamic software updating for C. In *Proc of the ACM SIGPLAN conference on Programming language design and implementation* (2006), PLDI '06, pp. 72–83.

[7] Österberg, D., and Lilius, J. Rethinking software updating: Concepts for improved updatability. Tech. Rep. 550, Turku Centre for Computer Science, Sep 2003.

[8] Seifzadeh, H., Abolhassani, H., and Moshkenani, M. S. A survey of dynamic software updating. *Journal of Software: Evolution and Process* (2012).

[9] Stoyle, G. P. *A Theory of Dynamic Software Updates*. PhD thesis, University of Cambridge, 2006.